cited 10 references.

```
     (FILE 'USPAT' ENTERED AT 08:22:12 ON 11 APR 1999)
L1         14761 S 395/?/CCLS
L2           624 S L1 AND ((INTERRUPT? OR EXCEPTION?) (5A) HANDLER?)
L3            77 S L2 AND (STACK? (P) SUBROUTINE? (P) RETURN?)
L4             1 S L3 AND JAVA
L5           733 S L1 AND (INTERRUPT? OR EXCEPTION?)/AB
L6            47 S L5 AND (STACK? (P) SUBROUTINE? (P) RETURN?)
L7             3 S L5 AND (STACK? (5A) STORE? (5A) SUBROUTINE? (5A) RETURN?
)
L8             6 S L1 AND (MISSILE?)/AB
L9             0 S L8 AND NAVY/AS
L10            0 S NAVY?/AS AND L8

=> s l4 and (store? (5a) pointer? (5a) stack?)

          516427 STORE?
           38773 POINTER?
          167653 STACK?
             385 STORE? (5A) POINTER? (5A) STACK?
L11            0 L4 AND (STORE? (5A) POINTER? (5A) STACK?)

=> s l4 and (store? (5a) stack?)

          516427 STORE?
          167653 STACK?
            6297 STORE? (5A) STACK?
L12            1 L4 AND (STORE? (5A) STACK?)

=> s l4 and (store? (5a) frame? (5a) stack?)

          516427 STORE?
          450003 FRAME?
          167653 STACK?
             125 STORE? (5A) FRAME? (5A) STACK?
L13            0 L4 AND (STORE? (5A) FRAME? (5A) STACK?)

=> s l5 and (interrupt? (5a) stack?)

          223073 INTERRUPT?
          167653 STACK?
            1553 INTERRUPT? (5A) STACK?
L14           86 L5 AND (INTERRUPT? (5A) STACK?)

=> s l14 (stack? (5a) store? (5a) pointer?)

MISSING OPERATOR 'L14 (STACK?'

=> s l14 and (stack? (5a) store? (5a) pointer?)

          167653 STACK?
          516427 STORE?
           38773 POINTER?
             386 STACK? (5A) STORE? (5A) POINTER?
L15           10 L14 AND (STACK? (5A) STORE? (5A) POINTER?)

=> d l15 kwic 1-10
```

US PAT NO: 5,822,606 [IMAGE AVAILABLE] L15: 1 of 10
US-CL-CURRENT: **395/800.16**; 712/23, 24, 203

ABSTRACT:
The . . . compiler and has two software stack pointers, one for the vector processors and one for the scalar processor, plus an **interrupt stack** pointer. The Parallel DSP Chip executes a single task in parallel. Using an enhanced C compiler, simple, familiar, scalar processing. . .

SUMMARY:

BSUM(111)

The . . . herein and has two software stack pointers, one for the vector processors and one for the scalar processor, plus an **interrupt stack** pointer.

DETDESC:

DETD(117)

The . . . are stored in memory at the address specified by the register in the Scalar Processor that is designated as the **Interrupt Stack** Pointer. Using a push-down stack, the register is decremented accordingly. Since memory is accessed by the Scalar Processor 107 and. . .

DETDESC:

DETD(131)
Interrupt Stack Pointer 621

DETDESC:

DETD(158)

Three . . . set of Vector Processors, and the Instruction Unit. All are implemented as registers in the Scalar Processor. The corresponding scalar **stack**, vector **stack** and **interrupt stack** are all software entities that are implemented in memory as 501 at a location chosen by the programmer, and can. . .

DETDESC:

DETD(162)

The **interrupt stack** pointer is provided for the pushing and popping of processor status during interrupts. Processor status comes from the Scalar Processor and the Instruction Unit. ~~**Interrupts** use the **interrupt stack pointer** to **store** processor status information so that a return from interrupt to the previously executing~~ code is possible. It is a self-incrementing. . .

DETDESC:

DETD(302)

A . . . routine instruction fetching. Three pushdown stacks can be used, one for scalar operands, one for vector operands and one for **interrupts**. The **stack** pointers for the scalar operands and the interrupts are implemented by register/counters so they can be updated without the use. . .

US PAT NO: 5,761,491 [IMAGE AVAILABLE] L15: 2 of 10
US-CL-CURRENT: **395/591**; 712/230

ABSTRACT:
A . . . hardware support for misaligned operands. Once the automatic alignment has occurred, the data processor stores a format field in an **exception** stack frame to indicate information about the alignment of the stack pointer at the time of the error. When the **exception** has been serviced, the processor uses the four bit format field **stored** in the **exception stack** frame to restore the stack **pointer** to its original value at the time of the **exception**.

DETDESC:

DETD(2)

The . . . has been serviced and a return from exception (RTE) instruction is executed, the processor uses the four bit format field **stored** in the exception **stack** frame to restore the stack **pointer** to its original value at the time of the exception.

DETDESC:

DETD(14)

An . . . "B" signal. Additionally, register file 228 provides an SP(1:0) signal. The SP(1:0) signal provides bits zero and one of a **Stack Pointer** value **stored** in register file 228. Multiplexer 2 230 is coupled to register A 234 and multiplexer 3 232 is coupled to. . .

DETDESC:

DETD(17)

Stack . . . bits define the byte address within a thirty-two bit wide address space. The SP(1) signal provides bit one of a **Stack Pointer** value **stored** in **stack pointer** 244 of register file 228. Similarly, the SP(0) signal provides bit zero of a **stack pointer** value **stored** in **stack pointer** 244 of register file 228. The SP(1) signal is provided to a first input of AND gate 272 and the. . .

DETDESC:

DETD(34)

While . . . constant, -4, is then transferred to multiplexer 232 and subsequently stored in register B 236. At this point, the adjusted **stack pointer** value is **stored** in register A 234 and the constant, -4, is stored in register B 236. The adjusted stack pointer value corresponds. . .

DETDESC:

DETD(57)

The . . . stack pointer value at the time of the exception. The ALU output is gated onto the Execute Result signal and **stored** in **stack pointer** register 244 of register file 228 as the current stack pointer.

DETDESC:

DETD(66)

In . . . has been serviced and a return from exception (RTE) instruction is executed, the processor uses the four bit format field **stored** in the exception **stack** frame to restore the stack **pointer** to its original value at the time of the exception.

CLAIMS:

CLMS(9)

9. A data processor, comprising:
control means for generating a first data processor **interrupt** signal;
a **stack** pointer register for storing and providing a current address value having a first portion and a second portion;
a logic circuit coupled to the control means for receiving the first **interrupt** signal and coupled to the **stack** pointer for receiving a first portion of the current address value, the logic circuit creating an adjusted address value; and
a. . .

US PAT NO: 5,640,548 [IMAGE AVAILABLE] L15: 3 of 10
US-CL-CURRENT: **395/561**; 712/202, 244

ABSTRACT:
A . . . system (100). In one form, the present invention is a more time efficient solution to the problem of unstacking and **stacking** registers (154-158) during **interrupt** processing in a data processing system (100). By taking advantage of the fact that pulling a register value off of. . . the values stored in the memory which is being used as the stack, the present invention reduces the unstacking and **stacking** each time that two **interrupts** are processed back to back with no non-**interrupt** processing in between. The present invention eliminates the unstacking of the program counter register (158) and the restacking of registers. . .

SUMMARY:

BSUM(9)

Some . . . family of microprocessors, available from Motorola, Inc. of Austin, Tex.), merely stack all of the register values, except for the **stack pointer** register value. The value **stored** in the **stack pointer** register is not stacked because it is used to continuously point to whichever entry in the **stack** is available next. The value **stored** in the **stack pointer** register is also called the "stack pointer".

SUMMARY:

BSUM(16)

In the present MC68HC11 family of microprocessors, the **stack** is most commonly used for **interrupt** processing, for subroutine calls, and for temporary storage of data values. However, regardless of the reason for a stacking operation. . .

SUMMARY:

BSUM(20)

In . . . of registers. The present invention also includes the step of determining whether to accept a first interrupt. If the first **interrupt** is accepted, a **stack** pointer value is changed without performing any **stacking**. If the first **interrupt** is not accepted,

a second portion of the plurality of registers is unstacked.

DRAWING DESC:

DRWD(2)

FIG. 1 illustrates, in flow diagram form, a representation of prior art register stacking and **stacking** which is performed during **interrupt** processing in the present MC68HC11 family of microprocessors;

DETDESC:

DETD(2)

A way was needed to reduce the amount of time spent **stacking** and unstacking registers for **interrupt** processing. It was also necessary for the new approach to be software compatible with the present MC68HC11 family of microprocessors.

DETDESC:

DETD(3)

The present invention significantly reduces the amount of time spent **stacking** and unstacking registers for **interrupt** processing. In the present MC68HC11 family of microprocessors, the "Return From Interrupt" (RTI) instruction is used by programmers as the. . . software interrupt service routine. The RTI instruction performs the function of unstacking all of the registers which had been previously **stacked** due to the **interrupt**.

DETDESC:

DETD(22)

Control . . . shown. Control logic 170 is coupled to stacking and unstacking logic circuitry 172. Control logic 170 is also coupled to **interrupt** logic circuitry 174. **Stacking** and unstacking logic 172 is bi-directionally coupled to bus 160. Interrupt logic 174 is also bi-directionally coupled to bus 160.

DETDESC:

DETD(25)

The present invention reduces the amount of time spent **stacking** and unstacking registers for **interrupt** processing. Each time that two interrupts are processed back to back with no non-interrupt processing in between, the present invention. . .

DETDESC:

DETD(29)

Referring . . . another interrupt is taken, (i.e. following the "yes" paths from diamonds 20 and 21), then registers 154-158 are once again **stacked** (see rectangle 11). If another **interrupt** is not taken, (i.e. the "no" path from either diamond 20 or diamond 21), then the flow returns to continue. . .

DETDESC:

DETD(37)

The present invention bypasses the need for unstacking the program counter register 158 and the need for re-**stacking** registers 154-157 if another **interrupt** is immediately taken. Instead of re-stacking registers 154-157, the stack pointer is adjusted by the proper amount so the stack. . .

DETDESC:

DETD(56)

The . . . is in a first mode or a second mode. If data processing system 100 is in the first mode, the **stacking** and unstacking of registers during **interrupt** processing occurs in the same manner as in the present MC68HC11 family of microprocessors, that is, in the same manner as illustrated in FIG. 1. If data processing system 100 is in the second mode, the **stacking** and unstacking of registers during **interrupt** processing occurs in accordance with the present invention, that is, in the same manner as illustrated in FIG. 2 and. . .

CLAIMS:

CLMS(1)

We . . .
(A) unstacking a first portion of the plurality of registers;
(B) determining whether to accept a first interrupt;
(C) if the first **interrupt** is accepted, changing a **stack** pointer value without performing any **stacking**; and
(D) if the first **interrupt** is not accepted, unstacking a second portion of the plurality of registers.

CLAIMS:

CLMS(7)

7. . . .
1) unstacking a first portion of a plurality of registers;
2) determining whether to accept a first interrupt;
3) if the first **interrupt** is accepted, changing a **stack** pointer value without performing any **stacking**; and
4) if the first **interrupt** is not accepted, unstacking a remaining portion of the plurality of registers.

CLAIMS:

CLMS(11)

11. . . . of registers in a data processing system during interrupt processing, the method comprising the steps of:
(A) accepting a first **interrupt**;
(B) **stacking** the plurality of registers;
(C) concurrently with said stacking step (B), incrementally adjusting a stack pointer value so that the stack. . . the stack pointer value having a second stack pointer value after completion of said unstacking step (E);
(G) accepting a second **interrupt**;
(H) changing the **stack** pointer value to the first stack pointer value without performing any stacking; and
(I) beginning execution of a second interrupt service. . .

CLAIMS:

CLMS(12)

12. . . .

stack, the stack pointer value having the second stack pointer value after completion of said step (J);
(L) accepting a third **interrupt**; and
(M) changing the **stack** pointer value to the first stack pointer value without performing any stacking.

CLAIMS:

CLMS(22)

22. . . .
registers;
a stack pointer register having a stack pointer value;
a memory stack, coupled to said plurality of registers and to said **stack** pointer;
interrupt determining means for determining whether to accept an interrupt; and
control means for selectively changing said stack pointer value, said control means being coupled to said **stack** pointer register and to said **interrupt** determining means, said control means changing the stack pointer value without performing any **stacking** if said **interrupt** determining means accepts the interrupt.

US PAT NO: 5,634,046 [IMAGE AVAILABLE] L15: 4 of 10
US-CL-CURRENT: **395/568**; 712/202

ABSTRACT:
The . . . pointer register in a computer is made available for general purpose use by programs executing at lower privilege levels than **interrupt** handlers. A set of instructions in such programs, excluding **stack** operations, **stores** data other than the **stack pointer**, such as operands, and the like, in the stack pointer register. When switching execution to an **interrupt** handler on an **interrupt**, return address data for the currently executing program is pushed onto a **stack** at the **interrupt** handler's privilege level. Thus, storing other data in the stack **pointer** register does not result in **stack** corruption. Also, these instructions can **store** data in a scratch portion of a stack segment beyond the current stack pointer.

SUMMARY:

BSUM(8)

In . . . occurs. Specifically, when the microprocessor receives an interrupt, the address of the instruction currently being executed is pushed onto the **stack**. The starting address of an **interrupt** handler (a program for servicing an interrupt) is then loaded into the instruction register from a predetermined location in main. . .

SUMMARY:

BSUM(9)

In . . . any time. Accordingly, if data other than the stack pointer (i.e. the address of the last data pushed onto the **stack**) is **stored** into the **stack pointer** register, a subsequent **interrupt** can cause the current instruction address to be stored at other than the next sequential location in the stack segment,. . .

SUMMARY:

BSUM(11)

The . . . drawbacks of the prior art. According to the invention, a

program is executed at a lower privilege level than an **interrupt** handler to avoid **stack** corruption when using the stack pointer register for general purposes. Some computers, such as those having the Intel microprocessors and. . .

SUMMARY:

BSUM(12)

In . . . interrupt handler's privilege level on receiving an interrupt. The return address of the executing program is not pushed onto the **stack** until after switching to the **interrupt** handler's privilege level. Accordingly, the return address is stored in the **stack** segment of the **interrupt** handler's privilege level and not that of the currently executing program.

SUMMARY:

BSUM(13)

According to one aspect of the invention, a group of one or more operations in a program **stores** data other than the **stack pointer** in the stack **pointer** register. The stack pointer register is used as an additional general purpose register, for example, to store operands and results. . . of operations, such as in a critical loop of a program. Prior to executing the group of operations, the stack **pointer** contained in the **stack pointer** register is **stored** to memory. The **stack pointer** is later restored to the stack pointer register from memory after executing the operations. To avoid stack corruption, the program. . . Because the microprocessor switches to the interrupt handler's privilege level before the program's current return address is stored on the **stack**, **interrupts** occurring between the storing and restoring of the stack pointer do not result in stack corruption.

SUMMARY:

BSUM(14)

According . . . a group of operations in a program stores data in the stack segment beyond the memory location indicated by the **stack pointer** currently **stored** in the **stack pointer** register. This part of the stack segment beyond the current stack pointer can serve as a "scratch" space for temporary. . .

DETDESC:

DETD(9)

As . . . rule, operands and pointers to operands in memory, other than the stack pointer (to the current top 54 of the **stack** 52), are not **stored** in the **stack pointer** register 37. The reason for this rule is that the microprocessor 12 utilizes the contents of the stack pointer register. . . 12 interprets the contents of the stack pointer register 37 as the stack pointer. Accordingly, the microprocessor 12 decrements the **stack pointer** register 37, and **stores** data at the location indicated by the register. When performing a pop operation, the microprocessor 12 reads the data at. . . indicated by the contents of the stack pointer register 37, and increments the register. If data other than the current **stack pointer** is **stored** in the **stack pointer** register 37, a subsequent push operation will store data somewhere in the stack segment, possibly overwriting valid data in the. . .

DETDESC:

DETD(10)

Even though a program itself may not include push and pop stack operations, the microprocessor 12 performs these **stack** operations when an **interrupt** is received. When an interrupt is received the microprocessor transfers program execution to another program referred to herein as an. . .

DETDESC:

DETD(14)

In accordance with the invention, an application program executing on the computer 10 (FIG. 1) uses the stack **pointer** register 37 (FIG. 2) to **store** data other than the **stack pointer** itself, such as operands and pointers to operands, without potentially causing stack corruption. Stack corruption is avoided by assigning the. . . performing any push operations which store the program's return address data. Also, when changing stack segments, the microprocessor 12 first **stores** the current contents of the **stack pointer** and stack segment registers 37, 39 in the application program's task state segment 20 before loading these registers with the. . .

DETDESC:

DETD(17)

As . . . instructions directing the microprocessor 12 to perform a stack operation. As explained above, stack operations performed while other data is **stored** in the **stack pointer** register 37 can potentially corrupt the stack 50.

DETDESC:

DETD(18)

In an initial step 84 of the method 80, the microprocessor 12 executing the instructions **stores** the **stack pointer** contained in the stack **pointer** register 37 to a location in the memory 14, such as in the data segment 25 associated with the program. This preserves the current stack pointer for later restoration. The microprocessor 12 can then proceed, at step 86, to **store** other data in the **stack pointer** register 37. At this point, the stack pointer register 37 can be used for general data storage just as any. . . may occur at any time during performance of the method 80), the application program's return address is pushed onto the **interrupt** handler's **stack** segment 21 at the **interrupt** handler's privilege level 60 due to the interrupt handler's assignment to a higher privilege level than the application program. Further,. . .

DETDESC:

DETD(20)

If . . . used for push and pop operations from the stack segment 26 at the application program's privilege level 63 to the **stack** segment 21 at the **interrupt** handler's privilege level 60. This switch takes place before a return address for the application program is stored with a. . .

DETDESC:

DETD(21)

With reference to FIGS. 3 and 7, the storage of a program's return address in the **stack** segment 21 at the **interrupt** handler's privilege level 60 by the microprocessor 12 also makes it possible for the application program to utilize a portion. . . pointer contained in the stack pointer register 37, because it may be overwritten if additional data is pushed onto the **stack** such as during an **interrupt**.

CLAIMS:

CLMS(2)

2. . . .
or more operations for storing data other than the stack pointer in the stack pointer register, the operations excluding any **stack** operations;
whereby on receiving an **interrupt** in executing the application program between the steps of storing and restoring the stack pointer, the processor switches to the. . . the first privilege level, the method thereby avoiding corruption of the stack allocated to the second privilege level when the **stack pointer** register **stores** data other than the **stack pointer** and an **interrupt** occurs.

CLAIMS:

CLMS(6)

6. . . .
a stack memory beyond a location of a current top stack element for data storage and retrieval, the operations excluding **stack** operations;
whereby on receiving an **interrupt** in executing the program between the steps of storing and restoring the stack pointer, the processor switches to the first. . . the first privilege level, the method thereby avoiding corruption of the stack allocated to the second privilege level when the **stack pointer** register **stores** data other than the **stack pointer** and an **interrupt** occurs.

CLAIMS:

CLMS(11)

11. . . .
the stack memory previously containing the top stack element at a time prior to the parameters being popped from the **stack** memory;
whereby on receiving an **interrupt** in executing the program between the steps of storing and restoring the stack pointer, the processor switches to the first. . . the first privilege level, the method thereby avoiding corruption of the stack allocated to the second privilege level when the **stack pointer** register **stores** data other than the **stack pointer** and an **interrupt** occurs.

CLAIMS:

CLMS(12)

12. . . .
the first privilege level, the apparatus thereby avoiding corruption of the stack allocated to the second privilege level when the **stack pointer** register **stores** data other than the **stack pointer** and an **interrupt** occurs.

CLAIMS:

CLMS(13)

13. . . .
or more operations for storing data other than the stack pointer in the stack pointer register, the operations excluding any **stack** operations;

whereby on receiving an **interrupt** in executing the application program between the steps of storing and restoring the stack pointer, the processor switches to the. . . the first privilege level, the method thereby avoiding corruption of the stack allocated to the second privilege level when the **stack pointer** register **stores** data other than the **stack pointer** and an **interrupt** occurs.

US PAT NO: 5,475,822 [IMAGE AVAILABLE] L15: 5 of 10
US-CL-CURRENT: **395/569**; 364/259.9, 263.2, 285.1, 285.2, DIG.1

ABSTRACT:
The . . . When a program counter (72) points to a first instruction byte, a first data processing operation is initiated. If an **interrupt** occurs during execution of the first data processing operation, intermediate data calculations held in a plurality of temporary registers (64,. . . decremented to point to the first instruction byte and the instruction continues executing the data processing operation as though no **interrupt** occurred.

DETDESC:

DETD(39)

When . . . TEMP1 64, TEMP2 66, and TEMP3 68 are stored at an address in internal memory 32 pointed to by a **stack pointer** value **stored** in **stack pointer** register 72. The stack **pointer** value is decremented and the denominator stored in TEMP1 64 is stored at the memory location pointed to by the. . . to point to a next memory location at which a first portion of the numerator stored in TEMP2 66 is **stored**. Again, the **stack pointer** value is decremented and the second portion of the numerator stored in TEMP3 68 is stored at the memory location. . .

DETDESC:

DETD(42)

The . . . 32. The second portion of the numerator is transferred via Internal Data bus 34 and A bus 74 to be **stored** in TEMP3 68. The **stack pointer** is then incremented to access the first portion of the numerator from internal memory 32. The first portion of the numerator is transferred via Internal Data bus 34 and A bus 74 to be **stored** in TEMP2 66. Again, the **stack pointer** is incremented to access the denominator from internal memory 32. The denominator is transferred by Internal Data bus 34 and. . .

DETDESC:

DETD(50)

If, . . . of TEMP1 64, TEMP2 66, and TEMP3 68 are stacked in internal memory 32 at a location specified by the **stack pointer stored** in **stack pointer** register 72. The steps of storing the contents of TEMP3 68, TEMP2 66, and TEMP1 64 require 4 timing cycles.

DETDESC:

DETD(54)

The . . . in a plurality of temporary registers, is saved in the

stack memory at a location pointed to by the value **stored** in **stack pointer** register 72. Additionally, the program counter is incremented by one so that it points to a second byte of the. . .

CLAIMS:

CLMS(1)

We . . .
instruction byte when the interrupt signal is asserted;
xv) storing the first intermediate value and the second intermediate value in a **stack** memory;
xvi) executing an **interrupt** routine in response to the interrupt signal;
xvii) retrieving the first intermediate value and the second intermediate value from the stack. . .

CLAIMS:

CLMS(6)

6. . . .
second instruction byte when the interrupt signal is asserted;
storing the first intermediate value and the second intermediate value in a **stack** memory; and
executing an **interrupt** routine in response to the interrupt signal.

CLAIMS:

CLMS(18)

18. . . .
second instruction byte when the interrupt signal is asserted;
storing the first intermediate value and the second intermediate value in a **stack** memory; and
executing an **interrupt** routine in response to the interrupt signal.

US PAT NO: 5,440,747 [IMAGE AVAILABLE] L15: 6 of 10
US-CL-CURRENT: 710/267; **395/500**; 710/22, 59, 266; 713/323

ABSTRACT:
A data processor that includes an **interrupt** controller, a condition code register, a condition code stacking register, a data memory with a stacking area, a data processing unit, and an **interrupt** request decoder. The **interrupt** controller receives **interrupt** request signals including a break **interrupt** request signal and a plurality of standard **interrupt** request signals. The condition code register stores condition code values including a data processor mode control value. The data processing. . . mode" value and are enabled when the data processor mode control value is set to a predefined "run mode". The **interrupt** decoder responds to a received break **interrupt** request signal by generating: (A) first, condition code register control signals to copy the mode control value stored in the. . . code register control signals to set the mode control value to-the predefined "run mode" value. It responds to a standard **interrupt** request signal by generating: (A) first, condition code register control signals to set the mode control value to the predefined. . .

DETDESC:

DETD(128)

Furthermore, . . . address in the stack pointer register 1816 (sp) incremented by one. Thus, this address points to the address in the **stacking** area where the **interrupt** return address is located.

DETDESC:

DETD(136)

Furthermore, . . . time, the address contents of the second stack pointer register 1817 (sp+1) are incremented so that the new address it **stores** is the address in the **stack pointer** register 1816 (sp) incremented by one. Thus, this address points to the next address for a pop operation.

DETDESC:

DETD(138)

In . . . described, a program control stacking register 1513 (pcstck) and a condition code stacking register 1697 (ccrstck) were employed for break **interrupt** requests in order to **stack** the contents of the program control register 1505 (pc3) and the condition code register 1674 (ccr) during the break mode. However, for the other standard **interrupt** requests, the **stack** pointers 1816 (sp) and 1817 (sp+1) of the address unit (AU) 1800 were used for stacking the contents of these. . .

DETDESC:

DETD(139)

However, . . . stacking the contents of the program control register 1503 (pc3) and the condition code register 1674 (ccr) for the break **interrupt** requests as well. Alternatively, the **stack** pointer registers 1816 (sp) and 1817 (sp+1) can be used with the condition code stacking register 1697 (ccrstck) so that. . .

CLAIMS:

CLMS(4)

4. . . .
signal by generating control signals to copy said first instruction address from said second program counter register to said program **stacking** register;
said **interrupt** decoder state machine responding to a predefined break return instruction by generating (A) control signals to copy said first instruction. . . signal by generating control signals to copy said first instruction address from said second program counter register to said data **stacking** area; and
said **interrupt** decoder state machine responding to a predefined standard interrupt return instruction by generating (A) control signals to copy said first. . .

US PAT NO: 5,274,817 [IMAGE AVAILABLE] L15: 7 of 10
US-CL-CURRENT: **395/670**; 364/222.4, 232.8, 238.3, 244, 244.3, 247, 247.7, 251, 251.3, 259, 259.2, 261.3, 280, 281.3, DIG.1; 712/233, 244

ABSTRACT:
A . . . subroutine entry address code; passing control to the return address if the compared values are equal; and executing a software **interrupt** if the compared values are not equal.

DETDESC:

DETD(8)

For . . . is common in the art. A stack pointer (SP) 40 is provided for pointing to the highest location of data **stored** in the **stack** 38. The **stack pointer** 40 is a 16-bit register in the control unit 18 whose contents determine the location where data is to be. . .

DETDESC:

DETD(12)

FIG. . . . SW-L. Initially, the stack pointer 40 points to the top of the stack. When the 16 bit signature word is **stored** on the **stack** 38, the **stack pointer** 40 is decremented two positions so that it points to the first byte of the signature word or SW-H.

DETDESC:

DETD(16)

It . . . in block 110 will be answered in the negative because the signature word will not have been stored on the **stack** 38. The **interrupt** routine will be responsively executed, thereby preventing further harm due to the erroneous change in the program counter 38.

US PAT NO: 5,247,628 [IMAGE AVAILABLE] L15: 8 of 10
US-CL-CURRENT: **395/390**; 710/260; 712/244; 714/17

ABSTRACT:
A . . . irrespective of the location of the instructions in the sequence. The control circuitry includes the capability to receive an instruction **interrupt** signal. The control circuitry then determines which instruction generated the instruction **interrupt**. Upon this determination, the control circuitry resets the processors and the dispatching apparatus to the state that existed when the instruction that generated the instruction **interrupt** was earlier executed in order to re-execute the instruction that caused the **interrupt** signal in accordance with its location in the instruction sequence.

DETDESC:

DETD(6)

FIG. . . . 44 and 46. Also, the condition register 26 is connected via lines 48 and 50 to the condition register backup **stack** 34. The **interrupt** address register 28 is connected via line 70 to an adder 75. The adder 75 produces the interrupt address for. . .

DETDESC:

DETD(15)

FIG. . . . 24, the count register 22, or the condition register 26 to be updated, the old values of these registers are **stored** in the appropriate backup **stacks** and the respective tail **pointers** are incremented. There is a head and tail pointer for each backup stack. These pointers operate the three backup register. . .

US PAT NO: 4,491,912 [IMAGE AVAILABLE] L15: 9 of 10
US-CL-CURRENT: **395/590**; 364/244, 244.3, 244.6, 247, 247.2, 247.7, 247.8, 251, 251.1, 251.3, 259.9, 262.4, 262.5, 262.7, 262.8, 263.2, DIG.1; 710/261; 712/245

ABSTRACT:
A . . . microsubroutine to restore the return address of the microinstruction from said stack unit; a second control unit for monitoring an **interrupt** request; a second storage for saving therein

the content of said stack unit; a status register having a field for indicating the acceptance of the **interrupt** request in the course of the execution of the microprogram, and a third control unit responsive to the detection of the **interrupt** request by the second control unit in the course of the execution of the microprogram to indicate the acceptance of the **interrupt** request by the status register and save the content of the stack unit in said second storage, and responsive to the presence of the indication in the status register indicating the acceptance of the **interrupt** request at the end of the execution of an **interrupt** processing program to restore the content of the second storage to said stack unit.

SUMMARY:

BSUM(11)

In . . . is characterized by means for saving and retrieving the contents of a microinstruction address register, a stack pointer and a **stack** area, a halfway **interrupt** indicating flag additionally provided in a status register, means for checking an interrupt detection signal in the course of the. . .

DETDESC:

DETD(69)

As . . . stack area 52. Under the control of the program stored at the specific address and the following addresses, the information **stored** in the **stack pointer** 51 and the stack area 52 can be saved in the main storage 43.

CLAIMS:

CLMS(1)

What . . .
of an interrupt to said status register and said program counter;
third storage means for storing therein the contends of said **stack** means;
means including a halfway **interrupt** indicating field for indicating that an interrupt request was accepted in the course of the execution of a macro instruction;. . .

US PAT NO: 4,399,507 [IMAGE AVAILABLE] L15: 10 of 10
US-CL-CURRENT: **395/395**; 364/231.8, 232.8, 238, 244, 244.3, 245, 245.1, 251, 251.5, 254, 254.5, 254.8, 258, 258.1, 258.2, 259, 261.3, 261.4, 261.7, 262.4, 263, 263.1, 271.6, 271.7, 271.8, DIG.1; 369/200; 712/234

ABSTRACT:
An . . . fetch stage, the address of the next instruction to be returned to in the instruction storage is immediately available without **interrupting** the flow in the pipeline. A stack pointer in a stage of the pipeline between the instruction fetch stage and. . .

DETDESC:

DETD(13)

The . . . which affect the stack are: branch and stack unconditionally `BS,` return `RET,` and return and enable interrupts `RET ENABLE.` Unmasked **interrupts** also affect the **stack**.

DETDESC:

DETD(22)

A . . . operand of the last accessed instruction word in response to the decoding of the associated op code in the data **store** accessing decoder 22. A **stack pointer** 28 of FIG. 4 is in the second stage 2 and has a control input 25 connected to the data. . .

DETDESC:

DETD(81)

The . . . affect the Stack 10 are: Branch and Stack unconditionally `BS`, Return `RET`, and Return and Enable Interrupts `RET ENABLE`. Unmasked **Interrupts** also affect the **Stack** 10.

DETDESC:

DETD(85)

During the first machine cycle, the value stored in the stack register 18 will be S.sub.i and the value **stored** in the **stack pointer** 28 will be i. The instruction sequencing decoder 20 in the first stage 1 decodes a previous instruction during the. . .

DETDESC:

DETD(92)

The . . . No-Ops and 2N+1 of which are return instructions, with N=0 or any positive integer. Finally, the receipt of any unmasked **Interrupt** affects the **STACK** 10 by causing the appropriate return address of the interrupted program to be `pushed` into the STACK Register 18 prior to being written into Data **Store** 8 using the **Stack Pointer** 28 incremented by 1 (i.e. SP=SP+1) as the address. The Interrupt causes a branch to the appropriate Interrupt Entry Address. . .

CLAIMS:

CLMS(1)

Having . . .
operand of said last accessed instruction word in response to the decoding of the associated op code in said data **store** accessing decoder;
a **stack pointer** in said second stage having a control input connected to said data store accessing decoder and an output, for selectively. . .

CLAIMS:

CLMS(4)

4. . . .
operand of said last accessed instruction word in response to the decoding of the associated op code in said data **store** accessing decoder;
a **stack pointer** in said second stage having a control input connected to said data store accessing decoder and an output, for selectively. . .

CLAIMS:

CLMS(7)

7. . . .
for selectively generating a data store address in response to the decoding of said last op code in said data **store** accessing decoder;
a **stack pointer** in said second stage having a control input connected to said data store accessing decoder and an output, for selectively. . .

CLAIMS:

CLMS(10)

10. . . .
decoder;
selectively incrementing or decrementing a data store address value for accessing the last instruction address stored in said instruction address **stack** in said data **store** in a **stack pointer** in said second stage having a control input connected to said data store accessing decoder;
accessing a location in said data. . .

CLAIMS:

CLMS(13)

13. . . .
operand of said last accessed instruction word in response to the decoding of the associated op code in said data **store** accessing decoder;
a **stack pointer** in said second stage having a control input connected to said data store accessing decoder and an output, for selectively. . .

CLAIMS:

CLMS(16)

16. . . .
operand of said last accessed instruction word in response to the decoding of the associated op code in said data **store** accessing decoder;
a **stack pointer** in said second stage having a control input connected to said data stored accessing decoder and an output, for selectively. . .

CLAIMS:

CLMS(19)

19. . . .
for selectively generating a data store address in response to the decoding of said last op code in said data **store** accessing decoder;
a **stack pointer** in said second stage having a control input connected to said data store accessing decoder and an output, for selectively. . .

CLAIMS:

CLMS(22)

22. . . .
decoder;
selectively incrementing or decrementing a data store address value for accessing the last instruction address stored in said instruction address **stack** in said data **store** in a **stack pointer** in said second stage having a control input connected to said data store

accessing decoder;
accessing a location in said data. . .

> d l15 1-10

1. 5,822,606, Oct. 13, 1998, DSP having a plurality of like processors controlled in parallel by an instruction word, and a control processor also controlled by the instruction word; Steven G. Morton, **395/800.16**; 712/23, 24, 203 [IMAGE AVAILABLE]

2. 5,761,491, Jun. 2, 1998, Data processing system and method for storing and restoring a stack pointer; Joseph C. Circello, et al., **395/591**; 712/230 [IMAGE AVAILABLE]

3. 5,640,548, Jun. 17, 1997, Method and apparatus for unstacking registers in a data processing system; John A. Langan, et al., **395/561**; 712/202, 244 [IMAGE AVAILABLE]

4. 5,634,046, May 27, 1997, General purpose use of a stack pointer register; Amit Chatterjee, et al., **395/568**; 712/202 [IMAGE AVAILABLE]

5. 5,475,822, Dec. 12, 1995, Data processing system for resuming instruction execution after an interrupt and method therefor; James M. Sibigtroth, et al., **395/569**; 364/259.9, 263.2, 285.1, 285.2, DIG.1 [IMAGE AVAILABLE]

6. 5,440,747, Aug. 8, 1995, Data processor with control logic for storing operation mode status and associated method; Atsushi Kiuchi, 710/267; **395/500**; 710/22, 59, 266; 713/323 [IMAGE AVAILABLE]

7. 5,274,817, Dec. 28, 1993, Method for executing subroutine calls; Alan L. Stahl, **395/670**; 364/222.4, 232.8, 238.3, 244, 244.3, 247, 247.7, 251, 251.3, 259, 259.2, 261.3, 280, 281.3, DIG.1; 712/233, 244 [IMAGE AVAILABLE]

8. 5,247,628, Sep. 21, 1993, Parallel processor instruction dispatch apparatus with interrupt handler; Gregory F. Grohoski, **395/390**; 710/260; 712/244; 714/17 [IMAGE AVAILABLE]

9. 4,491,912, Jan. 1, 1985, Data processing system with improved microsubroutine facility; Masahiro Kainaga, et al., **395/590**; 364/244, 244.3, 244.6, 247, 247.2, 247.7, 247.8, 251, 251.1, 251.3, 259.9, 262.4, 262.5, 262.7, 262.8, 263.2, DIG.1; 710/261; 712/245 [IMAGE AVAILABLE]

10. 4,399,507, Aug. 16, 1983, Instruction address stack in the data memory of an instruction-pipelined processor; Michael R. Cosgrove, deceased, et al., **395/395**; 364/231.8, 232.8, 238, 244, 244.3, 245, 245.1, 251, 251.5, 254, 254.5, 254.8, 258, 258.1, 258.2, 259, 261.3, 261.4, 261.7, 262.4, 263, 263.1, 271.6, 271.7, 271.8, DIG.1; 369/200; 712/234 [IMAGE AVAILABLE]

```
  *            6:30am - 9:00pm Monday through Friday              *
  *            7:30am - 5:00pm Saturday, Sunday, Holidays         *
  *                                                               *
  *      APS is unavailable Thanksgiving Day, Christmas Day,      *
  *      and New Year's Day.                                      *
  *                                                               *
  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
FILE 'USPAT' ENTERED AT 08:22:12 ON 11 APR 1999

  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
  *                W E L C O M E   T O   T H E                    *
  *          U. S.   P A T E N T   T E X T   F I L E              *
  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

=> s 395/?/ccls

L1        14761 395/?/CCLS

=> s l1 and ((interrupt? or exception?) (5a) handler?)

         223073 INTERRUPT?
         150893 EXCEPTION?
           9094 HANDLER?
           2078 (INTERRUPT? OR EXCEPTION?) (5A) HANDLER?
L2          624 L1 AND ((INTERRUPT? OR EXCEPTION?) (5A) HANDLER?)

=> s l2 and (stack? (p) subroutine? (p) return?)

         167653 STACK?
          19276 SUBROUTINE?
         564602 RETURN?
           1097 STACK? (P) SUBROUTINE? (P) RETURN?
L3            77 L2 AND (STACK? (P) SUBROUTINE? (P) RETURN?)

=> s l3 and java

            380 JAVA
L4            1 L3 AND JAVA

=> d l4 kwic


US PAT NO:     5,740,441 [IMAGE AVAILABLE]              L4: 1 of 1
US-CL-CURRENT: 395/704, 705, 707

SUMMARY:

BSUM(8)

 The present invention verifies the integrity of computer programs
written in a bytecode language, commercialized as the JAVA bytecode
language, which uses a restricted set of data type specific bytecodes.
All the available source code bytecodes in the.  .  .

SUMMARY:

BSUM(11)

 The merger of stack and register status maps requires special handling
for the instructions associated with exception handlers and the
instructions associated with subroutine calls (including "finally"
instruction blocks that are executed via a subroutine call whenever a.  .
.

DETDESC:
```

DETD(17)

Referring . . . interpreted will result in a series of executable instructions. A listing of all the source code bytecode instructions in the **JAVA** instruction set is provided in Table 1. The **JAVA** bytecode instruction set is characterized by bytecode instructions that are data type specific. Specifically, the **JAVA** instruction set distinguishes the same basic operation on different primitive data types by designating separate opcodes. Accordingly, a plurality of. . . numbers), with each such bytecode being used to process only data of a corresponding distinct data type. In addition, the **JAVA** instruction set is notable for instructions not included. For instance, there are no instructions in the **JAVA** bytecode language for converting numbers into object references. These restrictions on the **JAVA** bytecode instruction set help to ensure that any bytecode program which utilizes data in a manner consistent with the data type specific instructions in the **JAVA** instruction set will not violate the integrity of a user's computer system.

DETDESC:

DETD(25)

During normal execution of programs using languages other than the **Java** bytecode language, the interpreter must continually monitor the operand stack for overflows (i.e., adding more data to the stack than. . .

DETDESC:

DETD(58)
a table of **exception handlers**.

DETDESC:

DETD(59)

Each entry in the **exception handlers** tables gives a start and end offset into the bytecodes, an exception type, and the offset of a **handler** for the **exception**. The entry indicates that if an exception of the indicated type occurs within the code indicated by the starting and ending offsets, a **handler** for the **exception** will be found at the given handler offset.

DETDESC:

DETD(83)
(C) all **exception handlers** for this instruction; and

DETDESC:

DETD(85)

It is noted that the last instruction of most **exception handlers** is a "goto"instruction. More generally, the successor instruction for the end of an **exception handler** is simply the successor instruction for the last instruction of the **exception handler**.

DETDESC:

DETD(89)

For instance, if a successor instruction is an **exception handler**, the Stack Status portion of the SnapShot of the successor

instruction is defined to contain a single object of the **exception** type indicated by the **exception handler** information (i.e., the stored data type for the first virtual stack element indicates the object type of the **exception handler**), and furthermore the stack counter of the SnapShot of the successor instruction is set to a value of 1.

DETDESC:

DETD(99)

Verification Considerations For **Exception Handlers**

DETDESC:

DETD(100)

An **exception handler** is a routine that protects a specified set of program code, called a protected code block. The **exception handler** is executed whenever the applicable **exception** gets thrown during execution of the corresponding protected code.

DETDESC:

DETD(101)

As indicated above, the Stack Status portion of the SnapShot for the first instruction of the **exception handler** contains a single object of the **exception** type indicated by the **exception handler** information (i.e., the stored data type for the first virtual stack element indicates the object type of the **exception handler**), and further more the stack counter of the SnapShot of the instruction is set to a value of 1.

DETDESC:

DETD(102)

The virtual register information of the SnapShot for the **exception handler**'s first instruction contains data type values only for registers whose use is consistent throughout the protected code, and contains "unknown". . .

DETDESC:

DETD(107)

In the **Java** bytecode language, the "finally" construct is implemented using the exception handling facilities, together with a "jsr" (jump to **subroutine**) instruction and "ret" (**return** from **subroutine**) instruction. The cleanup code is implemented as a **subroutine**. When it is called, the top item on the **stack** will be the **return** address; this **return** address is saved in a register. A "ret" is placed at the end of the cleanup code to **return** to whatever code called the cleanup.

DETDESC:

DETD(108)

To implement the "finally" feature, a special **exception handler** is set up for the protected code which catches all **exceptions**. This **exception handler**: (1) saves any **exception** that occurs in a register, (2) executes a "jsr" to the cleanup code, and (3) upon return from the cleanup. . .

DETDESC:

DETD(122)

One . . . in FIG. 4F. In addition, there may not be any uninitialized objects in a register in code protected by an **exception handler** or a finally code block. See steps 524, 526, 528 in FIG. 4F. Otherwise, a devious piece of code could. . . object when it had, in fact, initialized an object created in a previous pass through the loop. For example, an **exception handler** could be used to indirectly perform a backwards branch.

DETDESC:

DETD(130)

TABLE 1

BYTECODES IN **JAVA** LANGUAGE INSTRUCTION NAME	SHORT DESCRIPTION
nop	no operation
aconst.sub.-- null	push null object
iconst.sub.-- m1	push. . . dup top 2 elements. Skip one
dup2.sub.-- x2	dup top 2 elements. Skip two
swap	swap top two elements of **stack**.
iadd	integer add
ladd	long add
fadd	floating add
dadd	double float add
isub	integer subtract
lsub	long subtract
fsub. . .	
ifgt	goto if greater than
ifle	goto if less than or equal
if.sub.-- icmpeq	compare top two elements of **stack**
if.sub.-- icmpne	compare top two elements of **stack**
if.sub.-- icmplt	compare top two elements of **stack**
if.sub.-- icmpge	compare top two elements of **stack**
if.sub.-- icmpgt	compare top two elements of **stack**
if.sub.-- icmple	compare top two elements of **stack**
if.sub.-- acmpeq	compare top two objects of **stack**
if.sub.-- acmpne	compare top two objects of **stack**
goto	unconditional goto
jsr	jump **subroutine**
ret	**return** from **subroutine**
tableswitch	goto (case)
lookupswitch	goto (case)
ireturn	**return** integer from procedure
lreturn	**return** long from procedure
freturn	**return** float from procedure
dreturn	**return** double from procedure
areturn	**return** object from procedure
return	**return** (void) from procedure

```
getstatic       get static field value.
putstatic       assign static field value
getfield        get field value from object.
putfield.  .  . multidimensional array
ifnull          goto if null
ifnonnull       goto if not null
goto.sub.-- w   unconditional goto. 4byte offset
jsr.sub.-- w    jump subroutine. 4byte offset
breakpoint      call breakpoint handler
```

DETDESC:

DETD(131)

TABLE 2

Pseudocode for **JAVA** Bytecode Verifier

```
Receive Object Class File with one or more bytecode programs to
be verified.
/*  Perform initial checks.  .  .  not require inspection of bytecodes
    */
If file format of the class file is improper
Print appropriate error message
Return with Abort return code
}
If  (A)   any "final" class has a subclass;
    (B)   the class is a subclass of a "final".  .  .
          constant pool does not have a legal name, class and type
          signature
    {
Print appropriate error message
Return with Abort return code
}
For each Bytecode Method in the Class
{
/*    Data-flow analysis is performed on each method of the.  .  .  the
constant pool
      does not match the data type of the referenced constant pool
      item,
      (E) any exception handler does not have properly specified
      starting and ending points,
      {
      Print appropriate error message
      Return with Abort return code
      }
Create: status data structures: stack counter, stack status
array,
register status array, jsr bit vector array
Create SnapShot array with one SnapShot for every instruction in the
bytecode program
Initialize SnapShot for first instruction of program to indicate the
stack is empty and the registers are empty except for data types
indicated by the method's type signature (i.e., for arguments.  .  .
(e.g., in sequential order in program)
whose changed bit is set
Load SnapShot for the selected instruction (showing status of
stack and registers prior to execution of the selected
instruction) into the stack counter, virtual stack and the
virtual register array, and jsr bit vector array, respectively.
Turn off the selected instruction's changed bit
/*      Emulate the effect of this instruction on the stack and
        registers*/
Case(Instruction Type):
```

```
{
Case=Instruction pops data from Operand Stack
        {
        Pop operand data type information from Virtual Stack
Update Stack Counter
If Virtual Stack has Underflowed
        {
        Print error message identifying place in program that
          underflow occurred
        Abort Verification
        Return with abort return code
        }
Compare data type of each operand popped from virtual
        stack with data type required (if any) by the bytecode
        instruction
If type mismatch
        {
        Print message identifying place in program that data
          type mismatch occurred
        Set VerificationSuccess to False
        Return with abort return code
        }
}
Case=Instruction pushes data onto Operand Stack
{
Push data type information onto Virtual Stack
Update stack counter
If Virtual Stack has Overflowed
        {
        Print message identifying place in program that
          overflow occurred
        Set VerificationSuccess to False
        Return with abort return code
        }
}
Case=Instruction uses data stored in a register
{
If type mismatch
        {
        Print message.  .  .  new data type
If instruction places an uninitialized object in a register and
        the instruction is protected by any exception handler
        (including the special exception handler for a "finally"
        code block)
        {
        Print error message
        Set VerificationSuccess to False
        }
}
Case=Backwards Branch
{
If Virtual Stack or Virtual Register Array contain any
        uninitialized object data types
        {
        Print error message
        Set VerificationSuccess.  .  .  }
}
} /* EndCase */
/*    Update jsr bit vector array */
If the current instruction is in a subroutine that is the target of a
jsr
{
For each level of jsr applicable to the current instruction
{
Update.  .  .  of all successor instructions, including:
```

```
(A)     the next instruction if the current instruction is not an
        unconditional goto, a return, or a throw,
(B)     the target of a conditional or unconditional branch,
(C)     all exception handlers for this instruction,
(D)     when the current instruction is a return instruction, the
        successor instructions are the instructions immediately
        following all jsr's that target the called subroutine.
If the program can "fall off" the last instruction
{
Set VerificationSuccess to False
Return with Abort return code value
}
/*  Merge the stack counter, virtual stack, virtual register
array and
    jsr bit
vector arrays into the SnapShots of each of the successor
instructions */
Do for each successor instruction:
{
If the successor instruction is the first instruction of an exception
handler,
{
Change the Stack Status portion of the SnapShot of the
        successor instruction to contain a single object of the
        exception type indicated by the exception handler
        information.
Set stack counter of the SnapShot of the successor
        instruction to 1.
Performs steps noted below for successor instruction
        handling.  .  .  array.
}
If this is the first time the SnapShot for a successor instruction
has been visited
{
Copy the stack counter, virtual stack, virtual register array
        and jsr bit vector array into the SnapShot for the
        successor instruction
Set the changed bit for the successor instruction
}
Else      /* the instruction has been visited before */
{
If the stack counter in the Status Array does not match the
        stack counter in the existing SnapShot, or the two
        stacks are not identical with regard to data types
        (except for differently typed object handles)
        {
        Set VerificationSuccess to False
        Return with Abort return code value
        }
Merge the Virtual Stack and Virtual Register Array values
        into the values of the existing SnapShot:
        (A) if two corresponding stack elements or two
        corresponding register elements contain different
        object handles, replace the specified data type for the
        stack or register element with the closest common
        ancestor of the two handle types;
        (B) if two corresponding register.  .  .
        current instruction is a "ret" instruction:
          1)  for any register that the bit vector indicates
              that the subroutine has accessed or modified,
              use the data type of the register at the time of
              the return, and
          2)  for other registers, use the data type of the
              register at the time of the preceding jsr
              instruction.
```

```
        /*  Note that return, break and continue instructions
            inside a code block protected by a "finally"
            exception handler are treated the same as a "jsr"
            instruction (for a subroutine call to the "finally"
            exception handler) for verification purposes. */
        Copy the jsr bit vectors into the SnapShot of the
          successor instructions only to the extent that those
          successor instructions are inside the same
          subroutines as the current instruction.
        Set the changed bit for each successor instruction for
          which the merging of the stack and register values
          caused any change to the successor instruction's
          SnapShot.
        }
}         /* End of Do.  .  .  */
}         /* End of Do Loop for Instruction Emulation */
}         /* End of Loop for Bytecode Methods */
Return (VerificationSuccess)
```

```
=> d his


     (FILE 'USPAT' ENTERED AT 08:22:12 ON 11 APR 1999)
L1         14761 S 395/?/CCLS
L2           624 S L1 AND ((INTERRUPT? OR EXCEPTION?) (5A) HANDLER?)
L3            77 S L2 AND (STACK? (P) SUBROUTINE? (P) RETURN?)
L4             1 S L3 AND JAVA

=> s l1 and (interrupt? or exception?)/ab

          19309 INTERRUPT?/AB
           3053 EXCEPTION?/AB
L5          733 L1 AND (INTERRUPT? OR EXCEPTION?)/AB

=> s l5 and (stack? (p) subroutine? (p) return?)

         167653 STACK?
          19276 SUBROUTINE?
         564602 RETURN?
           1097 STACK? (P) SUBROUTINE? (P) RETURN?
L6           47 L5 AND (STACK? (P) SUBROUTINE? (P) RETURN?)

=> s l5 and (stack? (5a) store? (5a) subroutine? (5a) return?)

         167653 STACK?
         516427 STORE?
          19276 SUBROUTINE?
         564602 RETURN?
             55 STACK? (5A) STORE? (5A) SUBROUTINE? (5A) RETURN?
L7            3 L5 AND (STACK? (5A) STORE? (5A) SUBROUTINE? (5A) RETURN?)

=> d l7 kwic 1-3


US PAT NO:      5,838,950 [IMAGE AVAILABLE]              L7: 1 of 3
US-CL-CURRENT: 395/500; 710/5

ABSTRACT:
The . . . supports many features found in traditional add-in card SCSI
host adapters. These features include bus master transfers, fast/wide
SCSI, one interrupt per command, scatter/gather, overlapped seeks,
tagged queuing, etc.

DETDESC:
```

DETD(161)

. . .

to subroutine

	at address in next address field (push next address + 1 onto stack for the **return**)
RET	Performs an unconditional **subroutine return** to the address **stored** on the top of **stack**
NB	No branch

US PAT NO: 4,755,935 [IMAGE AVAILABLE] L7: 2 of 3
US-CL-CURRENT: **395/580**; 364/DIG.1; 711/213

ABSTRACT:
A . . . the jump instruction, if not already in the buffer, is loaded into the buffer.
Special jump instructions facilitate subroutine calls and **interrupts** by allowing jumps to be executed to target instructions which are not at the beginning of a track. A CPU/memory-system. . . under the control of the CPU, one or more of which may be used to control conditional jumps and signal **interrupts** to the memory system.

DETDESC:

DETD(43)

The . . . nested subroutines are employed in a program, each current track register value corresponding to the instruction to be executed on **return** from a **subroutine** is **stored** on a **stack** in the central processing unit memory used for storing data. In the multiple context embodiment of the present invention, the. . .

US PAT NO: 4,488,227 [IMAGE AVAILABLE] L7: 3 of 3
US-CL-CURRENT: **395/591**; 364/232.8, 238, 238.6, 238.7, 241.2, 241.3, 244, 244.3, 247, 247.2, 261.3, 262, 262.4, 262.8, 263.2, DIG.1; 710/262, 264, 269

ABSTRACT:
A computer system which facilitates the execution of nested subroutines and **interrupts** is disclosed. As each branch transfer within the program is executed by a control area logic, a microcommand initiates the. . . previously stored return addresses. Thus, a sequential return to unfinished routines or subroutines is provided. When the subroutine or hardware **interrupt** service routine is completed, a code in the address field enables the return address of the previously branched from or **interrupted** routine to be retrieved from the first register in the push down stack and to provide it as the address. . . pops all other stored return addresses one level in the stack. In addition to providing multiple levels of subroutine and **interrupt** nesting, any number of subroutines or hardware **interrupts** may be partially completed since the last operating subroutine or hardware **interrupt** service routine is always the first one to be completed. Logic is also provided to detect the occurrence of a hardware **interrupt** during a return sequence such that the requirement to simultaneously push and pop the stack is properly handled.

DETDESC:

DETD(35)

Return . . . the incremented current address as is provided from ROS address history register 66 when the executing microprogram branches to a

subroutine. In addition, **return** address **stack** 70 **stores** the nominal next address output by address multiplexer 1 60 whenever a hardware interrupt occurs which vectors the execution of. . .

=> d his

```
     (FILE 'USPAT' ENTERED AT 08:22:12 ON 11 APR 1999)
L1         14761 S 395/?/CCLS
L2           624 S L1 AND ((INTERRUPT? OR EXCEPTION?) (5A) HANDLER?)
L3            77 S L2 AND (STACK? (P) SUBROUTINE? (P) RETURN?)
L4             1 S L3 AND JAVA
L5           733 S L1 AND (INTERRUPT? OR EXCEPTION?)/AB
L6            47 S L5 AND (STACK? (P) SUBROUTINE? (P) RETURN?)
L7             3 S L5 AND (STACK? (5A) STORE? (5A) SUBROUTINE? (5A) RETURN?
)
```

=> d l6 1-47

1. 5,838,950, Nov. 17, 1998, Method of operation of a host adapter integrated circuit; Byron Arlen Young, et al., **395/500**; 710/5 [IMAGE AVAILABLE]

2. 5,812,813, Sep. 22, 1998, Apparatus and method for of register changes during execution of a micro instruction tracking sequence; Glenn Henry, et al., **395/394**; 712/228 [IMAGE AVAILABLE]

3. 5,778,207, Jul. 7, 1998, Assisting operating-system interrupts using application-based processing; Allen Henry Simon, **395/376** [IMAGE AVAILABLE]

4. 5,751,985, May 12, 1998, Processor structure and method for tracking instruction status to maintain precise state; Gene W. Shen, et al., **395/394** [IMAGE AVAILABLE]

5. 5,734,857, Mar. 31, 1998, Program memory expansion using a special-function register; Gerald Gaubatz, 711/2; **395/500**; 709/1; 711/5; 712/230 [IMAGE AVAILABLE]

6. 5,717,903, Feb. 10, 1998, Method and appartus for emulating a peripheral device to allow device driver development before availability of the peripheral device; Thomas J. Bonola, **395/500** [IMAGE AVAILABLE]

7. 5,684,934, Nov. 4, 1997, Page repositioning for print job recovery; Weilin Chen, et al., **395/113** [IMAGE AVAILABLE]

8. 5,673,426, Sep. 30, 1997, Processor structure and method for tracking floating-point exceptions; Gene W. Shen, et al., **395/591**; 712/222 [IMAGE AVAILABLE]

9. 5,659,721, Aug. 19, 1997, Processor structure and method for checkpointing instructions to maintain precise state; Gene W. Shen, et al., **395/569**; 712/23 [IMAGE AVAILABLE]

10. 5,655,115, Aug. 5, 1997, Processor structure and method for watchpoint of plural simultaneous unresolved branch evaluation; Gene W. Shen, et al., **395/586**; 712/23 [IMAGE AVAILABLE]

11. 5,651,124, Jul. 22, 1997, Processor structure and method for aggressively scheduling long latency instructions including load/store instructions while maintaining precise state; Gene W. Shen, et al., **395/391**; 712/23, 244 [IMAGE AVAILABLE]

12. 5,649,136, Jul. 15, 1997, Processor structure and method for

maintaining and restoring precise state at any instruction boundary; Gene W. Shen, et al., **395/591**; 714/38 [IMAGE AVAILABLE]

13. 5,644,742, Jul. 1, 1997, Processor structure and method for a time-out checkpoint; Gene W. Shen, et al., **395/591**; 714/38 [IMAGE AVAILABLE]

14. 5,628,016, May 6, 1997, Systems and methods and implementing exception handling using exception registration records stored in stack memory; Peter Kukol, **395/704**, **705** [IMAGE AVAILABLE]

15. 5,590,358, Dec. 31, 1996, Processor with word-aligned branch target in a byte-oriented instruction set; Ori K. Mizrahi-Shalom, et al., **395/380**, **500** [IMAGE AVAILABLE]

16. 5,560,036, Sep. 24, 1996, Data processing having incircuit emulation function; Toyohiko Yoshida, **395/568**; 364/231.8, 243.2, 245.4, 260, DIG.1; 711/170 [IMAGE AVAILABLE]

17. 5,557,759, Sep. 17, 1996, Video processor with non-stalling interrupt service; Dwayne T. Crump, et al., **395/200.49**; 710/129 [IMAGE AVAILABLE]

18. 5,538,436, Jul. 23, 1996, Two-part memory card socket connector and related interrupt handler; John I. Garney, 439/270; 361/684, 754; 364/929.4, 929.61, 953.3, DIG.2; **395/500**; 439/122, 259, 347, 489, 924.1 [IMAGE AVAILABLE]

19. 5,502,827, Mar. 26, 1996, Pipelined data processor for floating point and integer operation with exception handling; Toyohiko Yoshida, **395/591**; 364/230, 231.8, 261.6, DIG.1; 712/23 [IMAGE AVAILABLE]

20. 5,500,809, Mar. 19, 1996, Microcomputer system provided with mechanism for controlling operation of program; Toshifumi Nakai, 702/176; 364/242, DIG.1; **395/704** [IMAGE AVAILABLE]

21. 5,475,687, Dec. 12, 1995, Network and intelligent cell for providing sensing, bidirectional communications and control; Armas C. Markkula, Jr., et al., **395/200.54**; 370/419; 714/4 [IMAGE AVAILABLE]

22. 5,442,799, Aug. 15, 1995, Digital signal processor with high speed multiplier means for double data input; Tokumichi Murakami, et al., **395/800.36**; 364/258.2, DIG.1; 708/626 [IMAGE AVAILABLE]

23. 5,440,757, Aug. 8, 1995, Data processor having multistage store buffer for processing exceptions; Toyohiko Yoshida, **395/569**; 364/230.2, 231.8, 261.6, DIG.1 [IMAGE AVAILABLE]

24. 5,390,305, Feb. 14, 1995, Information processing apparatus capable of executing exception at high speed; Masafumi Takahashi, et al., **395/591**; 364/230, 230.1, 230.2, DIG.1; 712/212, 227 [IMAGE AVAILABLE]

25. H 1,385, Dec. 6, 1994, High speed computer application specific integrated circuit; Karl D. Stickel, et al., **395/800.36**; 708/277, 653 [IMAGE AVAILABLE]

26. 5,274,817, Dec. 28, 1993, Method for executing subroutine calls; Alan L. Stahl, **395/670**; 364/222.4, 232.8, 238.3, 244, 244.3, 247, 247.7, 251, 251.3, 259, 259.2, 261.3, 280, 281.3, DIG.1; 712/233, 244 [IMAGE AVAILABLE]

27. 5,247,628, Sep. 21, 1993, Parallel processor instruction dispatch apparatus with interrupt handler; Gregory F. Grohoski, **395/390**; 710/260; 712/244; 714/17 [IMAGE AVAILABLE]

28. 5,193,156, Mar. 9, 1993, Data processor with pipeline which disables exception processing for non-taken branches; Toyohiko Yoshida, et al., **395/586**; 364/231.8, 239, 239.8, 240, 244, 244.3, 247, 247.7, 247.8, 251, 251.3, 255.1, 259, 259.9, 261.3, 261.5, 261.6, 261.9, 262.4, 262.8, 262.81, 262.9, 263.1, DIG.1; 712/244 [IMAGE AVAILABLE]

29. 5,182,811, Jan. 26, 1993, Exception, interrupt, and trap handling apparatus which fetches addressing and context data using a single instruction following an interrupt; Ken Sakamura, 710/264; 364/941, 941.1, 941.3, DIG.2; **395/500** [IMAGE AVAILABLE]

30. 5,161,247, Nov. 3, 1992, Digital signal processor matching data blocks against a reference block and replacing the reference block when a new minimum distortion block is calculated; Tokumichi Murakami, et al., **395/800.36**; 364/927.92, 927.93, 933.1, 933.3, 940, 942, 957, 957.1, 974, 974.1, 974.3, 975.2, DIG.2 [IMAGE AVAILABLE]

31. 4,841,439, Jun. 20, 1989, Method for restarting execution interrupted due to page fault in a data processing system; Atsuhiko Nishikawa, et al., **395/591**; 364/244, 244.3, 244.6, 246, 246.1, 251, 251.1, 251.2, 254, 254.3, 254.5, 261.3, 261.5, 262, 262.2, 262.4, 262.8, 262.9, 263.2, 265, 265.3, 266.3, 271.5, 271.6, 271.8, DIG.1; 714/17 [IMAGE AVAILABLE]

32. 4,777,587, Oct. 11, 1988, System for processing single-cycle branch instruction in a pipeline having relative, absolute, indirect and trap addresses; Brian W. Case, et al., **395/582**; 364/DIG.1 [IMAGE AVAILABLE]

33. 4,755,935, Jul. 5, 1988, Prefetch memory system having next-instruction buffer which stores target tracks of jumps prior to CPU access of instruction; Alan L. Davis, et al., **395/580**; 364/DIG.1; 711/213 [IMAGE AVAILABLE]

34. 4,750,112, Jun. 7, 1988, Data processing apparatus and method employing instruction pipelining; Walter A. Jones, et al., **395/393**; 364/DIG.1; 712/218, 238, 240 [IMAGE AVAILABLE]

35. 4,654,786, Mar. 31, 1987, Data processor using picosquencer to control execution of multi-instruction subroutines in a single fetch cycle; Michael J. Cochran, et al., **395/589**; 364/231.4, 231.6, 231.8, 238, 240, 240.2, 242.6, 242.7, 243, 243.1, 244, 244.6, 247, 247.2, 247.3, 247.4, 247.7, 261.3, 261.5, 261.6, 261.7, 262.4, 262.8, 262.9, 263, 263.1, 263.2, 280, 280.8, 281.3, DIG.1; 712/245 [IMAGE AVAILABLE]

36. 4,590,551, May 20, 1986, Memory control circuit for subsystem controller; Ronald D. Mathews, 711/150; 364/222.2, 228.1, 228.3, 228.5, 230, 230.4, 232.7, 238.4, 238.5, 240.8, 240.9, 251, 251.3, 252, 270.5, 270.6, 280, 280.4, 280.7, 280.8, 284, 284.2, 284.3, 284.4, DIG.1; **395/500** [IMAGE AVAILABLE]

37. 4,488,227, Dec. 11, 1984, Program counter stacking method and apparatus for nested subroutines and interrupts; Ming T. Miu, et al., **395/591**; 364/232.8, 238, 238.6, 238.7, 241.2, 241.3, 244, 244.3, 247, 247.2, 261.3, 262, 262.4, 262.8, 263.2, DIG.1; 710/262, 264, 269 [IMAGE AVAILABLE]

38. 4,438,492, Mar. 20, 1984, Interruptable microprogram controller for microcomputer systems; William J. Harmon, Jr., et al., **395/591**; 364/231.8, 231.9, 232.8, 232.9, 238, 238.4, 239, 239.4, 242.1, 244, 244.3, 244.6, 244.7, 244.9, 246, 246.3, 251, 251.1, 252, 254, 254.3, 255.1, 258, 259, 261.3, 261.5, 262, 262.1, 262.4, 262.5, 262.8, 263.1, 270.5, 270.6, 271, 271.1, 280, 280.8, 926.9, 927.8, 929.1, 931, 931.5, 933.2, 933.5, 933.6, 934, 937, 937.2, 938, 938.1, 938.2, 939, 939.7, 940, 941, 941.1, 942.7, 942.8, 943.9, 944.6, 946.2, 946.3, 946.6, 946.7, 947,

947.1, 948, 948.1, 948.34, 949, 949.1, 950, 950.1, 955, 957, 957.1, 957.3, 957.6, 958.5, 959, 960, 960.6, 964, 964.6, 965, 965.4, 965.5, 965.8, DIG.1, DIG.2 [IMAGE AVAILABLE]

39. 4,429,361, Jan. 31, 1984, Sequencer means for microprogrammed control unit; Tiziano Maccianti, et al., **395/569**; 364/221.9, 238.6, 238.8, 239, 239.4, 241.2, 241.3, 243, 243.6, 244, 244.6, 247, 247.2, 247.5, 247.6, 251, 251.1, 251.3, 254, 254.5, 259, 259.9, 261.3, 261.5, 262.4, 262.8, 263.2, 265, 265.6, 271.6, DIG.1; 712/243, 244, 248 [IMAGE AVAILABLE]

40. 4,399,507, Aug. 16, 1983, Instruction address stack in the data memory of an instruction-pipelined processor; Michael R. Cosgrove, deceased, et al., **395/395**; 364/231.8, 232.8, 238, 244, 244.3, 245, 245.1, 251, 251.5, 254, 254.5, 254.8, 258, 258.1, 258.2, 259, 261.3, 261.4, 261.7, 262.4, 263, 263.1, 271.6, 271.7, 271.8, DIG.1; 369/200; 712/234 [IMAGE AVAILABLE]

41. 4,398,244, Aug. 9, 1983, Interruptible microprogram sequencing unit and microprogrammed apparatus utilizing same; Paul Chu, et al., **395/591**; 364/232.8, 238, 239, 239.4, 242.1, 244, 244.3, 246, 246.2, 247, 247.1, 247.2, 247.8, 251, 251.1, 251.3, 261.3, 261.5, 262, 262.1, 262.4, 262.8, 264, 264.6, 280, 280.8, DIG.1; 712/228, 241 [IMAGE AVAILABLE]

42. 4,370,709, Jan. 25, 1983, Computer emulator with three segment microcode memory and two separate microcontrollers for operand derivation and execution phases; Robert E. Fosdick, **395/500**; 364/230.6, 231.4, 231.5, 231.6, 238.6, 238.9, 239, 239.6, 240, 240.2, 241.2, 241.6, 243, 243.7, 244, 244.6, 247, 247.1, 247.8, 251, 251.2, 258, 258.4, 262.4, 262.7, 262.8, 263.2, 270, 270.4, 271, 271.1, 578, DIG.1 [IMAGE AVAILABLE]

43. 4,326,247, Apr. 20, 1982, Architecture for data processor; George P. Chamberlin, **395/800.42**; 364/231.4, 231.7, 232.8, 232.9, 238.6, 238.7, 239, 239.4, 239.7, 239.9, 240, 240.1, 240.2, 242, 243, 243.2, 244, 244.3, 244.6, 247, 247.3, 247.4, 247.6, 258, 258.2, 258.3, 259, 259.2, 259.5, 259.7, 259.9, 261.3, 261.4, 264, 264.6, 271.6, 271.8, DIG.1 [IMAGE AVAILABLE]

44. 4,222,103, Sep. 9, 1980, Real time capture registers for data processor; George P. Chamberlin, **395/559**; 364/221.9, 222, 222.4, 232.8, 238.6, 238.7, 240, 240.1, 241.2, 241.3, 242.4, 243, 244, 244.3, 244.6, 247, 247.8, 252, 258, 258.2, 258.3, 259, 259.2, 259.5, 259.7, 261.3, 263.2, 270, 270.1, 280, 280.8, DIG.1; 377/20, 26; 710/262 [IMAGE AVAILABLE]

45. 4,200,930, Apr. 29, 1980, Adapter cluster module for data communications subsystem; Robert L. Rawlings, et al., **395/200.42**; 364/222.2, 228.1, 228.3, 228.4, 228.5, 229, 229.1, 230, 230.4, 232.7, 232.8, 238.2, 238.3, 238.4, 238.5, 239, 239.1, 239.3, 239.4, 239.6, 242.3, 242.5, 244, 244.6, 245.5, 245.6, 248.1, 259, 259.7, 268, 268.3, 284, 284.3, 284.4, DIG.1 [IMAGE AVAILABLE]

46. 4,107,785, Aug. 15, 1978, Programmable controller using microprocessor; William H. Seipp, **395/564**; 364/141, 921, 921.2, 926.9, 927.2, 927.8, 928, 929.2, 931, 931.1, 932, 932.6, 934, 934.4, 935, 935.2, 935.3, 936, 937, 940, 941, 941.1, 941.2, 941.3, 942.3, 942.8, 943.9, 944.6, 945.7, 946.2, 946.9, 947, 947.6, 948.1, 949, 950, 950.5, 951.5, 955, 955.2, 955.3, 955.4, 959.1, 961.1, 964, 964.3, 965, 965.5, DIG.2 [IMAGE AVAILABLE]

47. 3,891,974, Jun. 24, 1975, Data processing system having emulation capability for providing wait state simulation function; Brent W. Coulter, et al., **395/500**; 364/221, 221.2, 221.9, 231.4, 231.6, 232.3, 238.3, 243, 243.3, 243.4, 244, 244.3, 246, 246.1, 246.2, 246.3, 251,

251.5, 254, 254.5, 255.1, 255.2, 255.7, 256.3, 256.4, 262.4, 262.5, 263.2, 271, 271.2, 271.3, 271.5, DIG.1 [IMAGE AVAILABLE]

=> d l8 1-6

1. 5,719,797, Feb. 17, 1998, Simulator for smart munitions testing; Mark D. Sevachko, 364/578, 132, 164; **395/500**; 708/313, 315, 813, 819 [IMAGE AVAILABLE]

2. 4,740,909, Apr. 26, 1988, Real time data reduction system standard interface unit; Donald D. Conklin, et al., 710/68; 342/195; 364/922.5, 923.4, 927.92, 929.2, 931.4, 931.44, 933.8, 933.9, 939, 939.5, 944.9, 949.3, 965.77, DIG.2; **395/500**; 710/71 [IMAGE AVAILABLE]

3. 4,510,574, Apr. 9, 1985, Servosystem between a master actuator and a slave actuator; Jack Guittet, et al., **395/95**; 318/628; 414/5, 909; 901/2, 9, 50 [IMAGE AVAILABLE]

4. 4,099,229, Jul. 4, 1978, Variable architecture digital computer; H. Clifford Kancler, **395/380**; 364/223, 223.1, 232.7, 237.8, 238, 238.6, 238.7, 239, 239.2, 242.1, 242.4, 243, 243.3, 244, 244.1, 244.6, 251, 251.1, 251.3, 252.3, 252.6, 253, 253.1, 254.9, 259, 259.5, 260.4, 260.9, 262.4, 262.7, 262.8, 263.1, 267, 267.4, 270, 270.2, DIG.1; 712/241, 243, 245 [IMAGE AVAILABLE]

5. 4,090,250, May 16, 1978, Digital signal processor; Curtis E. Carlson, et al., **395/581**; 364/923.4, 926, 926.1, 926.2, 933.1, 933.3, 933.5, 937.1, 938, 938.3, 942.8, 947, 947.5, 947.6, 955, 957, 957.1, 959.1, 960, 960.2, 960.6, 964, 964.1, 965, 965.5, DIG.2; 712/245 [IMAGE AVAILABLE]

6. 4,037,202, Jul. 19, 1977, Microprogram controlled digital processor having addressable flip/flop section; John Terzian, **395/595**; 244/3.11; 364/221.7, 223, 223.1, 243, 243.3, 244, 244.6, 252, 260.4, 260.8, 262.4, 262.7, 262.8, 263, DIG.1 [IMAGE AVAILABLE]

=> d his

(FILE 'USPAT' ENTERED AT 08:22:12 ON 11 APR 1999)

```
L1         14761 S 395/?/CCLS
L2           624 S L1 AND ((INTERRUPT? OR EXCEPTION?) (5A) HANDLER?)
L3            77 S L2 AND (STACK? (P) SUBROUTINE? (P) RETURN?)
L4             1 S L3 AND JAVA
L5           733 S L1 AND (INTERRUPT? OR EXCEPTION?)/AB
L6            47 S L5 AND (STACK? (P) SUBROUTINE? (P) RETURN?)
L7             3 S L5 AND (STACK? (5A) STORE? (5A) SUBROUTINE? (5A) RETURN?
)
L8             6 S L1 AND (MISSILE?)/AB
L9             0 S L8 AND NAVY/AS
L10            0 S NAVY?/AS AND L8
```